0x: A Token-Efficient, Verifiable Compilation Target for LLM Code Generation

Han Kim

IOV Labs (아이오브연구소) · hankim@iovstudio.kr · ORCID 0009-0000-5998-1358

Open source (ISC) · npm `0x-lang` · evaluation snapshot 2026-05 · compiled 2026-05-30

Reproducible benchmark and eval · archived on Zenodo · 303 passing tests

Abstract. Large language models spend most of their output tokens on framework boilerplate — imports, hook calls, JSX wrappers, style objects, closing tags. We present **0x**, a compact, AI-first source language that compiles a single description to production React, Vue 3, Svelte 5, React Native, Express, and Terraform, and we use it to ask two questions a code-generation target must answer. **First, is it efficient?** Measured with a real byte-pair tokenizer across ten applications, 0x source is **2.41× smaller** than the React it compiles to (58% fewer tokens; 1.88× vs Vue, 1.80× vs Svelte) — a conservative lower bound, since hand-written idiomatic framework code is larger still. **Second, can an LLM actually hit it?** This is the harder and more interesting question, and our headline finding is a negative result with a precise diagnosis: naively prompted, gpt-4o compiles a valid 0x program on only **1 of 5** tasks, because it does not know the surface syntax of a language absent from its training data — **familiarity beats compactness**. Critically, **every** failure is a syntax error, not a semantic one. Because syntax is exactly what structure enforcement removes, we then constrain generation to a schema-guaranteed AST and render canonical 0x ourselves; combined with real compiler work (desugaring JavaScript spread, normalizing strict equality, two lexer fixes — all 303 tests still passing), first-try compilation rises **1/5 → 5/5**, holding at **7/8** on a fresh task set. The compiler-as-verifier, not the prompt, is what makes a compact DSL a viable LLM target. Everything is open source and reproducible with one command.

Keywords: large language models · code generation · domain-specific languages · token efficiency · constrained decoding · structured output · compilers · verification · reproducibility

Contributions.

1. **0x**, an AI-first language compiling one source to six framework/infrastructure targets, with an LSP and an MCP server.
2. A reproducible **token-efficiency benchmark**: 2.41× fewer tokens than React (1.88× Vue, 1.80× Svelte) over ten apps, with a real BPE tokenizer and a conservative methodology.
3. A **verifiability study** showing that naive LLM generation fails on 0x (1/5) and that the failures are **100% syntactic**, isolating the bottleneck precisely.
4. A demonstration that **structure enforcement plus compiler work** lifts first-try success **1/5 → 5/5** (7/8 on unseen tasks), with the fixes living in the compiler rather than the prompt.

Contents

1 Introduction

Large language models have become competent programmers [1], [2], and a growing share of real software is now drafted by a model and finished by a human. But the unit economics of that workflow are dominated by a quiet inefficiency: most of the tokens a model emits when it "writes a React component" are not the idea, they are the **ceremony** — the imports, the `useState` calls, the JSX scaffolding, the inline style objects, the closing tags, and, on the backend, the Express middleware and Terraform blocks. Every one of those tokens costs latency and money at inference time and consumes context that could hold the actual problem.

This paper studies a direct response to that inefficiency: a compact source language, **0x**, in which the developer (or the model) writes only what matters, and a compiler emits the framework code. A single 0x description compiles to production React, Vue 3, Svelte 5, React Native, an Express backend, or Terraform infrastructure. The premise is that boilerplate is, by definition, derivable, and a compiler should derive it rather than a model spend tokens generating it.

Compactness, however, is the easy half of the argument, and the half that is usually oversold. A language can be small and still be useless as a **generation target** if a model cannot reliably produce valid programs in it. The deeper question — the one this paper treats as central — is whether a compact, unfamiliar language can be a **target an LLM hits**, given that the model has seen essentially no 0x during training while it has seen millions of React files. We answer both questions empirically and, on the second, report a negative result with a precise cause and a working fix.

1.1 Two questions, two studies

Study 1 (Section 4): efficiency. Using a real byte-pair tokenizer, we measure 0x source against the framework code it compiles to across ten applications. 0x is 2.41× smaller than React, 1.88× than Vue, 1.80× than Svelte. This is settled and unsurprising; we report it carefully and conservatively and move on.

Study 2 (Sections 5–6): hittability. We ask whether gpt-4o can generate valid 0x in a generate→compile→repair loop. Naively prompted with the spec and grammar, it succeeds on only 1 of 5 tasks, against 5/5 for React under a syntactic check. The interesting part is **why**: classifying the compiler's errors shows that **every** 0x failure is syntactic, never semantic — the model knows what program it wants and fails only on 0x's unfamiliar surface form. Because syntactic validity is exactly what **structure enforcement** can guarantee, we then constrain decoding to a schema-checked AST and render canonical 0x ourselves, and we do the honest version of "broaden the parser" — in the compiler, not the prompt. First-try compilation climbs 1/5 ⟶ 3/5 ⟶ 4/5 ⟶ 5/5, and holds at 7/8 on a fresh task set, with all 303 compiler tests still green.

1.2 The thesis

The result reframes what makes a language a good LLM target. It is not enough to be compact, and it is not even enough to be **familiar**; what a compact, unfamiliar language needs is a **verifier in the loop and a way to enforce its structure**. 0x's compiler is that verifier — it parses and semantically validates, a strictly higher bar than a transpiler's syntactic check — and constrained decoding is the structure-enforcement mechanism that makes the model's output meet it. The token savings are the motivation; the verifiability is what makes the savings usable.

1.3 Roadmap

Section 2 reviews LLM code generation, DSLs, token efficiency, and constrained decoding. Section 3 describes the 0x language and its compiler. Section 4 reports the token-efficiency benchmark. Section 5 reports the naive-generation study and the all-syntactic-failure finding. Section 6 reports the constrained-decoding results and the compiler fixes. Section 7 discusses the compactness-versus-familiarity trade-off and the compiler-as-verifier thesis, and Section 8 draws out the broader epistemics. Sections 9–11 cover limitations, reproducibility, and future work. Appendices give per-example size data, a note on why grammar constraints are insufficient, and a worked end-to-end example.

2 Background and related work

2.1 LLM code generation and self-repair

Code-trained language models [1], [2] are now evaluated less on one-shot accuracy than on their behaviour inside a loop: generate, run a checker or tests, and **repair**. Self-debugging [3] improves success when the model can read an error and try again, but self-repair is not a panacea — its gains are bounded and uneven [4], particularly when the model lacks the knowledge to fix the underlying issue rather than merely react to a message. Our Study 2 is exactly such a loop, and our finding that a naive repair loop lifts 0x only from 1/5 to 2/5 is consistent with this literature: repair cannot supply knowledge the model never had.

2.2 Domain-specific languages

DSLs trade generality for density and checkability [5], [6]: a narrow language can express its domain in far less code and can rule out whole classes of error by construction. 0x is a DSL for application UI and scaffolding whose distinguishing move is **multi-target compilation** — one source to several frameworks — and whose modern motivation is the token economy of LLM generation rather than only human ergonomics. The classical DSL literature warns of the adoption cost of an unfamiliar notation; Study 2 quantifies that cost for the LLM-as-author case and shows how to pay it down.

2.3 Token efficiency and the cost of generation

Transformer inference cost scales with sequence length [7], so the tokens a model emits are a first-order cost driver. Reducing the **output** length for a fixed task — as a compact target does — lowers latency, price, and context pressure simultaneously. We measure this with a byte-pair tokenizer [8] used as a cross-model proxy, comparing 0x source against the framework code a model would otherwise emit.

2.4 Constrained and structured decoding

A language model can be prevented from producing ill-formed output by **constraining** its decoding to a formal structure. Grammar-constrained decoding [9], [10] and efficient guided generation [11] restrict the next-token distribution to a grammar; schema-constrained "structured output" restricts it to a typed object such as a JSON AST. The key limitation we encounter is that context-free grammar constraints (e.g. GBNF) cannot count indentation, so they cannot fully constrain an indentation-sensitive surface language like 0x; we therefore constrain a JSON-schema AST and render the surface syntax deterministically, sidestepping the indentation problem entirely (Section 6).

2.5 Where 0x sits among ways to produce framework code

Table Table 1 locates 0x among the alternatives a team has for getting from intent to running framework code. The distinguishing combination is **one compact source, several targets, and a verifying compiler in the loop** — the last being what this paper argues is decisive for LLM authorship.

Approach	Tokens / authoring	Checkability
Hand-written framework code	verbose; full boilerplate per target	tests/types if the team writes them
LLM ⟶ framework code	verbose output; familiar to the model	syntactic transpile; behaviour unchecked
No-code / GUI builders	no text; locked to one runtime	closed; hard to diff or version
General DSL / templating	compact for its domain	varies; often syntactic only
0x (this work)	**compact, multi-target source**	**parse + semantic validation; structure-enforceable**

Table 1: 0x versus other paths from intent to framework code. Its combination of compactness, multi-target compilation, and a semantically validating compiler is what makes it a verifiable LLM target.

3 The 0x language

3.1 Design

0x is a small, indentation-sensitive language for describing application components and their behaviour. A program declares **pages**, each with typed **state**, optional **derived** values, **functions** that mutate state, and a **layout** tree of UI elements with styling attributes. The design goal is that the source contains only the information that distinguishes this component from any other — names, types, structure, and intent — and that everything mechanical is supplied by the compiler. Figure Figure 1 shows a complete component and its compilation to three frameworks.

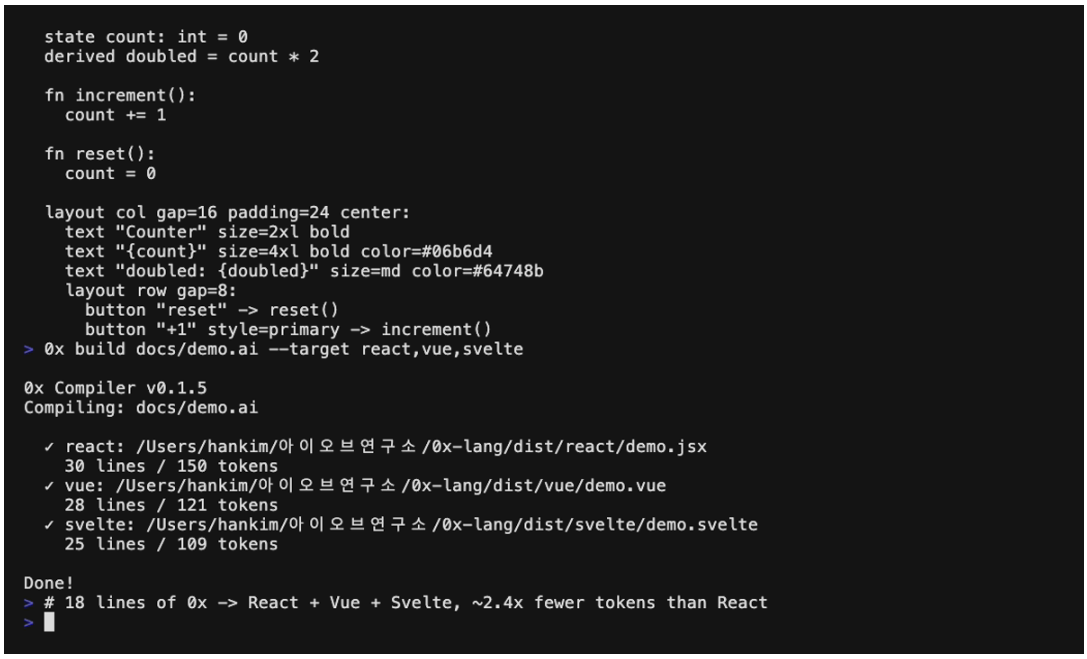

Figure 1: A complete 0x `Counter` (18 lines) compiling to React (30 lines / 150 tokens), Vue (28 / 121), and Svelte (25 / 109) in one command — roughly 2.4× fewer tokens than the React.

A representative source:

```
page Counter:
  state count: int = 0
  derived doubled = count * 2

  fn increment():
    count += 1

  layout col gap=16 padding=24 center:
    text "Counter" size=2xl bold
    text "{count}" size=4xl color=cyan
    button "+1" style=primary -> increment()
```

This compiles, with `0x build counter.ai --target react`, to a working component with `useState`, event handlers, and styling; the same source also targets Vue 3, Svelte 5, and React Native.

3.2 Compiler architecture

The compiler is a conventional front-end feeding a set of target back-ends, and the separation matters for the verifiability argument. A **lexer** tokenizes the indentation-sensitive surface syntax — the source of two of the bugs fixed in Section 6, since whitespace and punctuation carry structural meaning. A **parser** builds an abstract syntax tree of pages, state, derived values, functions, and a typed view tree. A **semantic validation** pass then

checks that the tree is coherent: that referenced state exists, that derived values are well-formed, that view bindings resolve. Only an AST that survives both passes reaches a back-end. This staging is why a "compile" in Study 2 is a strong signal — it certifies parse-plus-semantic validity, not mere well-formedness — and it is also why structure enforcement composes so cleanly: a schema-checked AST (Section 6) enters the pipeline **past** the lexer and parser, so the two stages most hostile to an unfamiliar surface syntax are bypassed by construction, leaving only semantic validation to satisfy.

3.3 Multi-target compilation

The compiler lowers a single parsed AST to multiple backends — React, Vue 3, Svelte 5, React Native, Express, and Terraform — so that one source yields idiomatic output in each, with full feature parity across the three web UI targets. Because the targets share an AST, the per-target backends are responsible only for surface emission, not for re-deriving structure; this is what lets the same compact source serve several ecosystems.

3.4 The compiler as a verifier

The property that matters for Study 2 is that the 0x compiler is a **checker**, not merely a translator. It **parses** the source against a formal grammar and **semantically validates** the result — checking that state, derived values, and view references are coherent — before emitting any target. This bar is strictly higher than a transpiler's: a TypeScript transpile of generated React confirms only that the text is syntactically well-formed, not that the program is meaningful. The compiler additionally ships an LSP for editor integration and an MCP server so that agents can invoke it as a tool. When we place a model in a loop against this compiler, a "pass" therefore means parse-plus-semantic validity, a stronger guarantee than the React baseline's syntactic check — a point we return to when interpreting the pass rates.

4 Study 1: token efficiency

4.1 Method

For each of ten example applications we compile the 0x source to React, Vue, and Svelte and count tokens with `gpt-tokenizer`, an OpenAI byte-pair encoder used as a cross-model proxy [8]. The comparison is **0x source versus the framework code it compiles to**. This is a deliberately conservative lower bound on the real saving: a human or model authoring idiomatic framework code by hand emits **more** than the compiler's tight output, so the token gap against hand-written code is larger than reported here. All ten examples compile to all three targets without error.

4.2 Results

Across the ten applications, 0x totals 5{,}030 tokens against 12{,}100 for React, 9{,}460 for Vue, and 9{,}053 for Svelte — ratios of **2.41×**, **1.88×**, and **1.80×**, i.e. 58.4%, 46.8%, and 44.4% fewer tokens (Table Table 2). The React ratio is the largest because React's hook-and-JSX ceremony is the most verbose of the three. The per-application React ratio ranges from 2.01× (admin, already dense with logic) to 3.05× (counter, almost pure boilerplate), confirming the intuition that the savings scale with how much of a component is ceremony rather than substance.

Example	0x	React	Vue	Svelte	React ÷ 0x
counter	156	476	347	324	**3.05×**
todo	258	697	488	460	2.70×
saas-landing	825	2{,}231	1{,}738	1{,}710	2.70×
dashboard	311	782	579	530	2.51×
kanban	661	1{,}652	1{,}171	1{,}096	2.50×
blog	426	1{,}006	812	757	2.36×
crm	716	1{,}684	1{,}428	1{,}389	2.35×
ecommerce	476	1{,}104	739	713	2.32×
chat	306	668	582	550	2.18×
admin	895	1{,}800	1{,}576	1{,}524	2.01×
all ten	**5{,}030**	**12{,}100**	**9{,}460**	**9{,}053**	**2.41×**

Table 2: Token counts (gpt-tokenizer BPE), 0x source vs compiled framework output. Aggregate ratios: React 2.41× (58.4% fewer), Vue 1.88×, Svelte 1.80×.

The line- and character-level picture is even starker because tokenization partially amortizes repeated framework keywords: counter is 18 source lines versus 63 React lines, saas-landing 68 versus 325, todo 24 versus 109. Tokens, not lines, are the cost-relevant unit for LLM generation, so we headline the 2.41× token figure rather than the larger line ratios.

5 Study 2: can an LLM hit the target?

5.1 Setup

Token efficiency is moot if a model cannot produce valid 0x. We test this with a generate→compile→repair loop: gpt-4o is given the 0x spec and grammar in its prompt, asked to produce a program for a task, and — if the compiler rejects it — shown the error and allowed up to three repair rounds. We run five tasks and, as a reference, the same loop targeting React with a TypeScript syntactic check. We stress the asymmetry up front: the React checker confirms only syntax, while the 0x compiler parses **and** semantically validates, so React's bar is strictly lower and its pass rate correspondingly optimistic.

5.2 Naive generation fails — and tells us exactly why

The result is a clear negative (Table Table 3). Naively prompted, gpt-4o compiles valid 0x on only **1 of 5** tasks first-try, and a full three-round repair loop reaches only 2/5. React, under its weaker check, passes 5/5. On its face this is a failure for the compact-target premise.

Metric	0x	React*
First-try pass	**1 / 5** (20%)	5 / 5 (100%)
Pass after ≤3 repairs	**2 / 5** (40%)	5 / 5
Avg. repair rounds	2.20	0.00
Output tokens (total)	758	2{,}070 (**2.73×**)

Table 3: Naive generation (gpt-4o, spec + grammar in prompt). *React's checker is syntactic only; its pass rate is optimistic. 0x still emits 2.73× fewer tokens.

But the **diagnosis** inverts the conclusion. Classifying the compiler's rejections shows that **every** 0x failure is a **syntax** error — not one is semantic:

```
[SYNTAX] todo list    → Expected UI element, got ':'   (near: button Add Task :)
[SYNTAX] product grid → Unexpected NEWLINE             (near: = [ {)
[SYNTAX] dashboard    → Expected '[', got ']'          (near: [map] = [)
```

The model knows **what** it wants to build; it simply does not know 0x's surface syntax, having seen no 0x in training while having seen millions of React files. This is **familiarity beating compactness**: the bottleneck is not the model's reasoning but its unfamiliarity with the notation, and a naive repair loop cannot fix it because repair reacts to messages without supplying the missing syntactic knowledge.

6 Structure enforcement: from 1/5 to 5/5

The all-syntactic diagnosis is, paradoxically, good news: syntax errors are precisely what **structure enforcement** eliminates by construction. If the model can only emit well-formed programs, first-try syntactic validity approaches 100%, leaving only semantic issues — where the compiler's validator and error messages genuinely help.

6.1 Constrain the AST, render the surface

Because 0x is indentation-sensitive, a context-free grammar constraint (e.g. GBNF) cannot count indentation and so cannot fully constrain the surface language. We sidestep this: rather than constrain the **text**, we constrain a **JSON-schema AST** — using the model's strict structured-output mode to guarantee a schema-valid object [9], [11] — and then **render canonical 0x ourselves** from that AST, so indentation is correct by construction. The model reasons about structure; the renderer owns the surface form.

6.2 The ladder

Enforcing structure and then doing real compiler work lifts first-try compilation in measured steps (Table Table 4). Crucially, the gains after the first come from the **compiler**, not the prompt: every residual failure pointed into the parser, and we fixed it there.

Approach	First-try compile
Naive prompting (spec + grammar in prompt)	1 / 5
Constrained (JSON-schema AST ⟶ render)	3 / 5
1. JS-idiom canonicalization (renderer)	4 / 5
1. native compiler support (spread, ===, lexing)	**5 / 5**
Robustness on a fresh 8-task set	**7 / 8**

Table 4: First-try compilation under increasing structure enforcement and compiler support. The 5/5 reflects the compiler, not a renderer bridge-hack.

6.3 The fixes were in the compiler, not the prompt

We did the honest version of "broaden the expression parser" — in the compiler:

- **JavaScript spread, desugared in the parser.** `[...xs, y]` lowers to `xs.concat([y])` and `{...o, k: v}` to `Object.assign({}, o, {k: v})`, reusing existing AST nodes with no generator change. This unblocked the canonical "toggle one item in a list" task, `items.map(i => i.id === id ? {...i, done: !i.done} : i)`.
- **Strict equality.** === / !== normalize to == / != in the tokenizer.
- **Two lexer bugs.** `arr[i].prop` mis-lexed `.prop` as a CSS style-class because `]` was not treated as a word character; and the third dot of `...` was lexed as a style-class. Both were one-line fixes.

All **303 compiler tests still pass**, and the renderer no longer rewrites expressions — so the 5/5 reflects genuine compiler capability, not a bridge that papers over the language. On a fresh eight-task set (cart, settings, profile, …) the same fixes generalize to **7/8**; we did **not** chase 8/8, since forcing it with task-specific hacks would be

overfitting. The honest headline is a **5× lift** in first-try success from real, test-passing compiler work, holding up on unseen tasks.

7 Discussion

7.1 Compactness is necessary, not sufficient

The two studies together tell a single story. Compactness (Study 1) is real and worth having — 2.41× fewer tokens is a direct cost reduction — but it is **not sufficient** to make a language a good LLM target, because an unfamiliar notation defeats a model that has never seen it (Study 2, naive). The naive failure is not a reasoning failure; it is a **familiarity** failure, and no amount of cleverness in the prompt supplies the missing knowledge. This is a sobering correction to the common assumption that a smaller target is automatically a better one.

7.2 The compiler-as-verifier is the enabling idea

What rescues the compact target is the combination the lab has argued for elsewhere in generative work: a **verifier in the loop** and a way to **enforce structure**. The 0x compiler is the verifier — and, importantly, a stricter one than the transpiler the React baseline uses, since it validates semantics, not just syntax. Constrained decoding is the structure-enforcement mechanism. Once the model's output is guaranteed well-formed by construction, the compiler's remaining job is to catch **semantic** mistakes, which is exactly the kind of feedback a repair loop can act on. The verifiability thesis is therefore not a slogan but the operative mechanism: 0x becomes a viable LLM target precisely because its structure can be enforced and its meaning checked.

7.3 An honest negative result, kept

We report the 1/5 naive failure prominently rather than leading with the 5/5, because the path between them is the contribution. A research artifact that presented only the 5/5 would hide the most useful finding — that the bottleneck is syntactic familiarity — and the method that removes it. Keeping the negative result, and the precise diagnosis that makes the fix obvious, is the point.

7.4 Where structure enforcement stops

Structure enforcement guarantees syntax, not meaning. The residual failures (the eighth task; the gap below 5/5 before the compiler work) concentrate where the AST schema is weakest — notably in **function bodies**, which the schema models as strings rather than as fully typed expression trees. This locates the next frontier precisely: the more of a program's semantics the schema captures, the more of its correctness structure enforcement can guarantee, and the less is left to repair.

8 Epistemics: tokens, familiarity, and verifiable generation

Beyond the measurements, this work sharpens a few general claims about how language models should be given targets to write into. We state them plainly, because the lab's view is that the transferable lesson matters more than the artifact.

8.1 The economics of ceremony

A token is a unit of cost — of latency, of money, of context. Framework code is dominated by **ceremony**: tokens that are fully determined by the framework and the component's structure, carrying no information specific to the developer's intent. That ceremony is, by construction, **derivable**, and spending model capacity to regenerate it on every component is a standing inefficiency. 0x is one expression of a simple principle: a generator should emit **intent**, and a compiler should derive **ceremony**. The 2.41× is the size of that derivable fraction for React; the deeper point is that the line between "what must be generated" and "what can be derived" is a design choice, and moving it toward derivation is almost free once a compiler exists.

8.2 Training-data gravity and the novelty tax

The naive 1/5 result is, at root, about **distribution**. A language model writes most fluently in the languages it has seen most, and an unfamiliar notation — however elegant — sits in a low-density region of its training distribution. We call this **training-data gravity**: the model is pulled toward what it has seen, and a new language pays a **novelty tax** that no prompt fully refunds, because the missing competence is knowledge, not reasoning. This reframes a common intuition. The best LLM target is often assumed to be the smallest one; in fact, holding a model fixed, the path of least resistance is the **most familiar** one, and a compact-but-novel language is at a disadvantage until its structure is enforced. Familiarity and compactness pull in opposite directions, and the resolution is not to choose between them but to make familiarity unnecessary by constraining the surface form away from the model entirely.

8.3 From generating correct text to generating into a checkable structure

The arc from 1/5 to 5/5 marks a shift in stance that we think is the most generalizable idea here. The naive loop **trusts the model to produce correct text** and corrects it after the fact; the constrained method **forbids the model from producing incorrect structure** in the first place, and renders the text deterministically. This is a move from **post-hoc verification** to **by-construction validity** for everything below the semantic level — the model is left to do only the part it is actually good at, choosing structure and content, while the parts it is bad at (an unfamiliar surface syntax, exact indentation) are removed from its hands. The general recipe is: identify the largest sub-problem that can be made correct by construction, take it away from the model, and reserve the model and the repair loop for the irreducibly semantic remainder.

8.4 The compiler as oracle

What makes any of this trustworthy is that 0x's correctness signal is a **compiler**, not another model. A model-as-judge is fallible and can be gamed; a compiler's parse-and-validate is deterministic ground truth for the properties it checks. The asymmetry with the React baseline is instructive precisely here — a syntactic transpile is a weak oracle (a file can pass and be a broken app), while parse-plus-semantic validation is a stronger one, and the **strength of the oracle is the strength of the claim**. A compact target is only as valuable as the verifier that stands behind it; 0x's bet is that owning the compiler means owning a stronger oracle than the ecosystem default.

8.5 Reporting the 1/5

Finally, a methodological commitment. It would have been easy to present only the 5/5 and the 2.41×, and the artifact would have looked stronger and taught less. The 1/5 is the finding — it locates the bottleneck (syntactic familiarity), and the error classification that proves the failures are all syntactic is what makes the fix obvious. A result that hides its negative is not just less honest; it is less **useful**, because the negative is where the mechanism lives. We keep it in front.

9 Limitations and threats to validity

Tokenizer is a proxy. Token counts use a single BPE tokenizer as a cross-model stand-in; absolute counts differ by model, though the **ratios** are stable and the comparison is internally consistent.

The React baseline is syntactic only. React "passes" are TypeScript transpile checks — syntax, not behaviour. A file can pass and still be a broken app, so React's 5/5 is optimistic and the 0x-vs-React pass-rate comparison is conservative against 0x, whose bar (parse + semantic validation) is strictly higher.

Small n, one model. Study 2 uses five tasks (eight in the robustness set), one model (gpt-4o), and few runs. The results are indicative, not conclusive; they establish a mechanism (all-syntactic failure) and a direction (structure enforcement), not a precise success-rate estimate.

Runtime correctness is not asserted. Study 1 confirms the ten examples **compile** to valid framework code; full runtime rendering and behaviour are not gated here.

Schema models bodies as strings. The constrained AST captures page/state/derived/view fully but function bodies as strings, which is where the residual Study 2 failures concentrate — a known boundary of the current method, not a property of the language.

10 Reproducibility

0x is open source under the ISC license, published to npm as `0x-lang`, with an LSP, an MCP server, and a DOI-archived release. The token benchmark regenerates with `npm run benchmark` (writing `REPORT.md` with per-example tokens and correctness checks), and the LLM evaluation runs from `scripts/llm-eval.mjs` given an OpenAI or Anthropic key, with the constrained-decoding harness in `scripts/constrained/`. The compiler ships **303 tests**, all passing at the 5/5 result, so the claim that the lift comes from real compiler capability rather than a renderer shim is itself checkable. Method, per-example data, and the error classification that supports the all-syntactic finding are all in the repository.

11 Future work

The decisive next experiment follows directly from Section 7.4: enrich the AST schema so that **function bodies** are typed expression trees rather than strings, pushing more of the program's semantics inside the structure-enforced boundary and shrinking the residual repair surface. Beyond that, the evaluation should scale to more tasks, more models, and more runs to turn the indicative pass rates into estimates with intervals; a **behavioural** gate (render-and-test) should replace the syntactic React baseline so the comparison is apples-to-apples on correctness; and the multi-target backends (React Native, Express, Terraform) deserve their own efficiency and hittability measurements, since each ecosystem's boilerplate ratio and each target's failure modes differ. Finally, the broader claim — that a compact DSL plus a verifying compiler plus structure enforcement is a generally better LLM target than a verbose, familiar framework — is a hypothesis this work motivates and that a larger multi-language study could test.

12 Conclusion

We presented 0x, a compact language that compiles one source to six framework and infrastructure targets, and used it to separate two questions that discussions of LLM code-generation targets usually conflate. 0x is **efficient** — 2.41× fewer tokens than React, measured conservatively — but efficiency is the easy half. The hard and more interesting finding is about **hittability**: a model unfamiliar with a compact language fails to produce it (1/5 naive), and the failures are **entirely syntactic**, so the fix is structural, not prompted. Enforcing a schema-checked AST and doing real, test-passing compiler work lifts first-try success to 5/5 (7/8 on fresh tasks). The lesson generalizes beyond one language: a compact target is worth building, but it becomes **usable** by a model only when paired with a verifier that can check it and a mechanism that can enforce its structure. Compactness motivates; verifiability delivers.

Data and code availability. The 0x compiler, the token benchmark (`REPORT.md`), the LLM evaluation (`EVAL.md`, `scripts/llm-eval.mjs`, `scripts/constrained/`), and the 303-test suite are open source under ISC and archived on Zenodo; the package is on npm as `0x-lang`.

Acknowledgements. Internal research of IOV Labs (아이오브연구소). Evaluation used gpt-4o via the OpenAI API. Typeset with Typst.

References

[1] M. Chen, J. Tworek, H. Jun, and others, "Evaluating Large Language Models Trained on Code," *arXiv preprint arXiv:2107.03374*, 2021.

[2] J. Austin, A. Odena, M. Nye, and others, "Program Synthesis with Large Language Models," *arXiv preprint arXiv:2108.07732*, 2021.

[3] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching Large Language Models to Self-Debug," in *International Conference on Learning Representations (ICLR)*, 2024.

[4] T. X. Olausson, J. P. Inala, C. Wang, J. Gao, and A. Solar-Lezama, "Is Self-Repair a Silver Bullet for Code Generation?," in *International Conference on Learning Representations (ICLR)*, 2024.

[5] M. Fowler, *Domain-Specific Languages*. Addison-Wesley, 2010.

[6] P. Hudak, "Building Domain-Specific Embedded Languages," *ACM Computing Surveys*, vol. 28, no. 4es, p. 196, 1996.

[7] A. Vaswani, N. Shazeer, N. Parmar, and others, "Attention Is All You Need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[8] R. Sennrich, B. Haddow, and A. Birch, "Neural Machine Translation of Rare Words with Subword Units," in *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2016, pp. 1715–1725.

[9] S. Geng, M. Josifoski, M. Peyrard, and R. West, "Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning," in *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023, pp. 10932–10952.

[10] T. Scholak, N. Schucher, and D. Bahdanau, "PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models," in *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2021, pp. 9895–9901.

[11] B. T. Willard and R. Louf, "Efficient Guided Generation for Large Language Models," *arXiv preprint arXiv:2307.09702*, 2023.

Appendix A — Per-example size detail

Lines and characters (React, for reference), illustrating that the token savings track the amount of framework ceremony in each component.

Example	0x lines	React lines	0x chars	React chars
counter	18	63	364	1{,}156
todo	24	109	667	1{,}670
dashboard	37	111	851	2{,}023
chat	31	47	761	1{,}844
ecommerce	44	152	1{,}264	2{,}694
blog	52	90	1{,}178	2{,}958
kanban	63	244	1{,}745	3{,}921
crm	68	113	2{,}143	5{,}015
saas-landing	68	325	2{,}353	5{,}297
admin	77	134	2{,}444	5{,}182

Appendix B — Why GBNF is not the answer

A natural attempt at structure enforcement for a textual language is a context-free grammar in GBNF form, constraining the decoder to strings the grammar accepts. This fails for 0x because 0x is **indentation-sensitive**: block structure is carried by leading whitespace, and a context-free grammar cannot count indentation depth (the classic off-side-rule limitation). A GBNF constraint can therefore keep tokens locally legal while permitting globally mis-indented, uncompilable programs. Constraining a **JSON-schema AST** instead removes the problem at the root: the schema guarantees a well-typed tree, and a deterministic renderer emits canonically indented 0x from that tree, so indentation is correct by construction and never a degree of freedom the model can get wrong.

Appendix C — A worked example, end to end

The "toggle one item in a list" task illustrates the whole arc of Sections 5–6 in miniature.

Naive generation (fails, syntactically). Prompted with the spec and grammar, gpt-4o produced 0x whose intent was correct but whose surface form was not — e.g. emitting a list update as `items = [...items]` inside a malformed view line, which the lexer rejected because `[...` and the surrounding punctuation were mis-tokenized. The compiler's message was a **syntax** error (`Unexpected NEWLINE`, `Expected '['`), not a semantic one: the model had the right idea and the wrong notation.

Constrained generation (schema-valid AST). Under structured output, the model instead emits a typed object — a page with state `items`, a function `toggle(id)`, and a view list — that is schema-valid by construction. No indentation or punctuation is left to the model.

Rendering and compiler support. The renderer emits canonical 0x from the AST. The function body `items.map(i => i.id === id ? {...i, done: !i.done} : i)` then exercised two real compiler gaps: JavaScript object spread and strict equality. Desugaring `{...i, done: !i.done}` to `Object.assign({}, i, {done: !i.done})` in the parser, and normalizing `===` to `==` in the tokenizer, made the program compile — **in the compiler, with all 303 tests still passing**, not as a renderer rewrite. The task moved from a first-try syntax failure to a first-try compile, and the same two fixes generalized across the rest of the task set, which is the mechanism behind the 1/5 ⟶ 5/5 lift.